

# MULTIPARTY TESTING PREORDERS

ROCCO DE NICOLA AND HERNÁN MELGRATTI

IMT Lucca School for Advanced Studies, Italy  
*e-mail address:* rocco.denicola@imtlucca.it

DC, FCEyN, Universidad de Buenos Aires - CONICET, Argentina  
*e-mail address:* hmelgra@dc.uba.ar

---

**ABSTRACT.** Variants of the must testing approach have been successfully applied in Service Oriented Computing for analysing the compliance between (contracts exposed by) clients and servers or, more generally, between two peers. It has however been argued that multiparty scenarios call for more permissive notions of compliance because partners usually do not have full coordination capabilities. We propose two new testing preorders, which are obtained by restricting the set of potential observers. For the first preorder, called uncoordinated, we allow only sets of parallel observers that use different parts of the interface of a given service and have no possibility of intercommunication. For the second preorder, that we call individualistic, we instead rely on parallel observers that perceive as silent all the actions that are not in the interface of interest. We have that the uncoordinated preorder is coarser than the classical must testing preorder and finer than the individualistic one. We also provide a characterisation in terms of decorated traces for both preorders: the uncoordinated preorder is defined in terms of must-sets and Mazurkiewicz traces while the individualistic one is described in terms of classes of filtered traces that only contain designated visible actions and must-sets.

## 1. INTRODUCTION

A desired property of communication-centered systems is the graceful termination of the partners involved in a multiparty interaction, i.e., every possible interaction among a set of communicating partners ends successfully, in the sense that there are no messages waiting forever to be sent, or sent messages which are never received. The theories of session types [THK94, HVK98] and of contracts [CGP08, CGP09, BZ07, LP07] are commonly used to ensure such kind of properties. The key idea behind both approaches is to associate to each process a type (or *contract*) that gives an abstract description of its external, visible behaviour and to use type checking to verify correctness of behaviours.

Services are often specified by sequential nondeterministic CCS processes [Mil89] describing the communications offered by peers, built-up from invoke and accept activities. These activities are abstractly represented either as input and output actions that take

---

*2012 ACM CCS: [Theory of computation]:* Models of computation—Concurrency—Process Calculi.

*Key words and phrases:* Must-testing preorder; Multiparty composition; Mazurkiewicz traces.

\* This is an extended and revised version of a paper with same title that appeared in the proceedings of the 10th International Symposium on Trustworthy Global Computing, Vol. 9533 of LNCS, Springer 2016.

place over a set of channels or as internal  $\tau$  actions. Basic actions can be composed sequentially (prefix operator “.”) or as alternatives (non deterministic choice “+”). Typically, the language for describing services does not have any operator for parallel composition. It is assumed that all possible interleavings are made explicit in the description of the service and communication is used only for modelling the interaction among different peers.

Services come equipped with a notion of *compliance* that characterises all valid clients of a service, i.e., those clients that are guaranteed to terminate after any possible interaction with the service. Compliance has been characterised by using a suitable variant of the must testing approach, which allows comparing processes according to the ability of external observers of distinguishing them. Processes that are must-equivalent are characterised by the set of tests or observers that they are able to pass: any observer is defined as a unique process that runs in parallel with the tested service and, consequently, all interactions with the observed service are handled by a unique, central process, i.e. the observer. Technically, two given processes  $p$  and  $q$  are related via the must preorder ( $p \sqsubseteq_{\text{must}} q$ ) if  $q$  passes all tests that are passed by  $p$ . Consequently,  $p$  and  $q$  are considered equivalent ( $p \approx_{\text{must}} q$ ) if they pass exactly the same tests.

If one considers a multiparty setting, each service may concurrently interact with several partners and its interface is often (logically) partitioned by allowing each partner to communicate only through dedicated parts of the interface. Also, in many scenarios, the peers of a specific service do not communicate with each other. In these situations, the classical testing approach to process equivalences or preorders turns out to unnecessarily discriminate services.

Consider the following scenario involving three partners: an organisation (the broker) that sells goods produced by a different company (the producer) to a specific customer (the client). The behaviour of the broker can be described by the following process:

$$B = \text{req}.\overline{\text{order}}.\overline{\text{inv}}.$$

The broker accepts requests on channel  $\text{req}$  and then places an order to the producer with the message  $\overline{\text{order}}$  and sends an invoice to the customer with the message  $\overline{\text{inv}}$ . In this scenario, the broker uses the channels  $\text{req}$  and  $\text{inv}$  to interact with the customer, while it interacts with the producer over the channel  $\text{order}$ . Moreover, the customer and the producer do not know each other and are completely independent. Hence, the order in which messages  $\overline{\text{order}}$  and  $\overline{\text{inv}}$  are sent is completely irrelevant for them. They would be equally happy with a broker defined as follows:

$$B' = \text{req}.\overline{\text{inv}}.\overline{\text{order}}.$$

Nevertheless, these two different implementations are not considered must-equivalent.

The main goal of this paper is to introduce alternative, less discriminating, preorders that take into account the distributed nature of the peers and their possibly limited coordination and interaction capabilities. A first preorder, called *uncoordinated must preorder*, is obtained by assuming that all clients of a given service do interact with it via fully disjoint sets of ports, i.e. they use different parts of its interface, have no possibility of intercommunication, and all of them terminate successfully in every possible interaction. It is however worth noting that these assumptions about the absence of communication among clients do not fully eliminate the possibility for clients of being influenced by other peers, in case one client does not behave as expected by the other. Due to this, it is possible to differentiate  $B$  from  $B'$  above when one of the peers refuses to synchronise over port  $\text{order}$ .

The second preorder, that we call *individualistic must preorder*, guarantees increased acceptability of offered services by allowing clients to take for granted the execution of

those actions of the service that are not explicitly of interest for them (i.e. not in their alphabet).

The two preorders are, as usual, defined in terms of the outcomes of experiments by specific sets of observers. For defining the uncoordinated must preorder, we allow only sets of parallel observers that cannot intercommunicate and do challenge services via disjoint parts of their interface. For defining the individualistic must preorder, we instead rely on parallel observers that, again, cannot intercommunicate but in addition perceive as silent all the actions that are not part of the interface of their interest. This is instrumental to avoid that a specific client recovers information about other involved peers. As expected, we have that the uncoordinated preorder is coarser than the classical must testing preorder and finer than the individualistic one.

Just like for classical testing preorders, we provide a characterisation for both new preorders in terms of decorated traces, which avoids dealing with universal quantifications over the set of observers whenever a specific relation between two processes has to be established. The alternative characterisations make it even more evident that our preorders permit action reordering. Indeed, the uncoordinated preorder is defined in terms of *Mazurkiewicz traces* [Maz95] while the individualistic one is described in terms of classes of traces quotiented via specific sets of visible actions. We would like to remark that our two preorders are different from those defined in [BZ08, Pad10, MYH09], which also permit action reordering by relying on buffered communication; additional details will be provided in §7.

**Synopsis** The remainder of this paper is organised as follows. In §2 we recall the basics of the classical must testing approach. In §3 and §4 we present the theory of uncoordinated and individualistic must testing preorders and their characterisation in terms of traces. In §5 we show that the uncoordinated preorder is coarser than the must testing preorder but finer than the individualistic one. In §6 we describe a Prolog implementation of the uncoordinated and individualistic preorders for the finite fragment of our specification language and use it for analysing a scenario involving a replicated data store. Finally, we discuss some related work and future developments in §7.

## 2. PROCESSES AND TESTING PREORDERS

Let  $\mathcal{N}$  be a countable set of action names, ranged over by  $a, b, \dots$ . As usual, we write co-names in  $\overline{\mathcal{N}}$  as  $\overline{a}, \overline{b}, \dots$  and assume  $\overline{\overline{a}} = a$ . We will use  $\alpha, \beta$  to range over  $\mathbf{Act} = (\mathcal{N} \cup \overline{\mathcal{N}})$ . Moreover, we consider a distinguished internal action  $\tau$  not in  $\mathbf{Act}$  and use  $\mu$  to range over  $\mathbf{Act} \cup \{\tau\}$ . We fix the language for defining services as the sequential fragment of CCS extended with a *success* operator, as specified by the following grammar.

$$p, q ::= \mathbf{0} \mid \mathbf{1} \mid \mu.p \mid p + q \mid X \mid \mathbf{rec}_X.p$$

The process  $\mathbf{0}$  stands for the terminated process,  $\mathbf{1}$  for the process that reports success and then terminates, and  $\mu.p$  for a service that executes  $\mu$  and then continues as  $p$ . Alternative behaviours are specified by terms of the form  $p + q$ , while recursive ones are introduced by terms like  $\mathbf{rec}_X.p$ . We sometimes omit trailing  $\mathbf{0}$  and write, e.g.,  $a.b + c$  instead of  $a.b.\mathbf{0} + c.\mathbf{0}$ . We write  $\mathbf{n}(p)$  for the set of names  $a \in \mathcal{N}$  such that either  $a$  or  $\overline{a}$  occur in  $p$ .

The operational semantics of processes is given in terms of a labelled transition system (LTS)  $p \xrightarrow{\lambda} q$  with  $\lambda \in \mathbf{Act} \cup \{\tau, \checkmark\}$ , where  $\checkmark$  signals the successful termination of an execution.

**Definition 2.1** (Transition relation). The transition relation on processes, noted  $\xrightarrow{\lambda}$ , is the least relation satisfying the following rules

$$\begin{array}{c} \mathbf{1} \xrightarrow{\checkmark} \mathbf{0} \quad \mu.p \xrightarrow{\mu} p \quad \frac{p \xrightarrow{\lambda} p'}{p + q \xrightarrow{\lambda} p'} \quad \frac{q \xrightarrow{\lambda} q'}{p + q \xrightarrow{\lambda} q'} \quad \frac{p[\mathbf{rec}_X.p/X] \xrightarrow{\lambda} p'}{\mathbf{rec}_X.p \xrightarrow{\lambda} p'} \end{array}$$

□

Multiparty applications, named *configurations*, are built by composing processes concurrently. Formally, configurations are given by the following grammar.

$$c, d ::= p \mid c \parallel d$$

We sometimes write  $\Pi_{i \in 0..n} p_i$  for the parallel composition  $p_0 \parallel \dots \parallel p_n$ . The operational semantics of configurations, which accounts for the communication between peers, is obtained by extending the LTS in Definition 2.1 with the following rules:

$$\frac{c \xrightarrow{\mu} c'}{c \parallel d \xrightarrow{\mu} c' \parallel d} \quad \frac{d \xrightarrow{\mu} d'}{c \parallel d \xrightarrow{\mu} c \parallel d'} \quad \frac{c \xrightarrow{\alpha} c' \quad d \xrightarrow{\bar{\alpha}} d'}{c \parallel d \xrightarrow{\tau} c' \parallel d'} \quad \frac{c \xrightarrow{\checkmark} c' \quad d \xrightarrow{\checkmark} d'}{c \parallel d \xrightarrow{\checkmark} c' \parallel d'}$$

All rules are standard apart for the last one that is not present in [DH84]. This rule states that the concurrent composition of processes can report success only when all processes in the composition do so.

We write  $c \xrightarrow{\lambda}$  when there exists  $c'$  such that  $c \xrightarrow{\lambda} c'$ ;  $\Rightarrow$  for the reflexive and transitive closure of  $\xrightarrow{\tau}$ ;  $c \xrightarrow{\lambda} c'$  for  $\lambda \in \mathbf{Act} \cup \{\checkmark\}$  and  $c \xRightarrow{\lambda} c'$ ;  $c \xRightarrow{\lambda_0 \dots \lambda_n} c'$  for  $c \xRightarrow{\lambda_0} \dots \xRightarrow{\lambda_n} c'$ , and  $c \xRightarrow{s}$  with  $s \in (\mathbf{Act} \cup \{\checkmark\})^*$  if there exists  $c'$  such that  $c \xRightarrow{s} c'$ . We write  $\mathbf{str}(c)$  and  $\mathbf{init}(c)$  to denote the sets of strings and enabled actions of  $c$ , defined as follows

$$\mathbf{str}(c) = \{s \in (\mathbf{Act} \cup \{\checkmark\})^* \mid c \xRightarrow{s}\} \quad \mathbf{init}(c) = \{\lambda \in \mathbf{Act} \cup \{\checkmark\} \mid c \xrightarrow{\lambda}\}$$

As behavioural semantics, we will consider the must-testing preorder [DH84]. We take all possible configurations as the set  $\mathcal{O}$  of *observers*, ranged over by  $o, o_0, \dots, o', \dots$ . If we do only allow observers to report success and use only sequential observers we recover the standard framework of [DH84].

**Definition 2.2** (must). A sequence of transitions  $p_0 \parallel o_0 \xrightarrow{\tau} \dots \xrightarrow{\tau} p_k \parallel o_k \xrightarrow{\tau} \dots$  is a maximal computation if either it is infinite or the last term  $p_n \parallel o_n$  is such that  $p_n \parallel o_n \not\xrightarrow{\tau}$ . Let  $p$  must  $o$  iff for each maximal computation  $p \parallel o = p_0 \parallel o_0 \xrightarrow{\tau} \dots \xrightarrow{\tau} p_k \parallel o_k \xrightarrow{\tau} \dots$  there exists  $j \geq 0$  such that  $o_j \xrightarrow{\checkmark}$ . □

We say that a computation  $c_o \xrightarrow{\mu_0} \dots c_i \xrightarrow{\mu_i} \dots \xrightarrow{\mu_n} c_{n+1}$  is unsuccessful when  $c_j \not\xrightarrow{\checkmark}$  for all  $0 \leq j \leq n+1$ , we say it successful otherwise.

The notion of passing a test represents the fact that a configuration, i.e., a set of partners, is able to successfully terminate every possible interaction with the process under test. Then, it is natural to compare processes accordingly to their capacity to satisfy partners.

**Definition 2.3** (must preorder).  $p \sqsubseteq_{\text{must}} q$  iff  $\forall o \in \mathcal{O} : p \text{ must } o \text{ implies } q \text{ must } o$ . We write  $p \approx_{\text{must}} q$  when both  $p \sqsubseteq_{\text{must}} q$  and  $q \sqsubseteq_{\text{must}} p$ . □

**2.1. Semantic characterisation.** The must testing preorder has been characterised in [DH84] in terms of the sequences of actions that a process may perform and the possible sets of actions that it may perform after executing a particular sequence of actions. This characterisation relies on a few auxiliary predicates and functions that are presented below. A process  $p$  *diverges*, written  $p \uparrow$ , when it exhibits an infinite, internal computation  $p \xrightarrow{\tau} p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} \dots$ . We say  $p$  *converges*, written  $p \Downarrow$ , otherwise. For  $s \in \text{Act}^*$ , the convergence predicate is inductively defined by the following rules:

- $p \Downarrow \epsilon$  if  $p \Downarrow$ .
- $p \Downarrow \alpha.s$  if  $p \Downarrow$  and  $p \xRightarrow{\alpha} p'$  implies  $p' \Downarrow s$ .

The *residual* of a process  $p$  (or a set of processes  $P$ ) after the execution of  $s \in \text{Act}^*$  is given by the following equations

- $(p \text{ after } s) = \{p' \mid p \xRightarrow{s} p'\}$ .
- $(P \text{ after } s) = \bigcup_{p \in P} (p \text{ after } s)$ .

**Definition 2.4** (Must-set). A *must-set* of process  $p$  (or set of processes  $P$ ) is  $L \subseteq \text{Act}$ , and  $L$  finite such that

- $p \text{ MUST } L$  iff  $\forall p'$  such that  $p \xRightarrow{\alpha} p'$ ,  $\exists \alpha \in L$  such that  $p' \xRightarrow{\alpha}$ .
- $P \text{ MUST } L$  iff  $\forall p \in P. p \text{ MUST } L$ . □

Then, the must testing preorder can be characterised in terms of strings and must-sets as follows.

**Definition 2.5.**  $p \preceq_{\text{must}} q$  if for every  $s \in \text{Act}^*$ , for all finite  $L \subseteq \text{Act}$ , if  $p \Downarrow s$  then

- $q \Downarrow s$ .
- $(p \text{ after } s) \text{ MUST } L$  implies  $(q \text{ after } s) \text{ MUST } L$ . □

**Theorem 2.6** ([DH84]).  $\sqsubseteq_{\text{must}} = \preceq_{\text{must}}$ .

### 3. A TESTING PREORDER WITH UNCOORDINATED OBSERVERS

The must testing preorder is defined in terms of the tests that each process is able to pass. Remarkably, the classical setting can be formulated by considering only sequential tests (see the characterisation of minimal tests in [DH84]). Each sequential test is a unique, centralised process that handles all the interaction with the service under test and, therefore, has a complete view of the externally observable behaviour of the service. For this reason, we refer to the classical must testing preorder as a *centralised preorder*. Multiparty interactions are generally structured in such a way that pairs of partners communicate through dedicated channels, for example, partner links in service oriented models or buffers in communicating machines [BBO12]. Conceptually, the interface (i.e., the set of channels) of a service is partitioned and the service interacts with each partner by using only specific sets of channels in its interface. In addition, there are scenarios in which partners frequently do not know each other and cannot communicate directly. As a direct consequence, the partners of a process cannot establish causal dependencies among actions that take place over different parts of the interface. These constraints reduce the discriminating power of partners and call for coarser equivalences that equate processes that cannot be distinguished by independent sets of sequential processes only interested in specific interactions.

**Example 3.1.** Consider the classical scenario for planning a trip. A user  $U$  interacts with a broker  $B$ , which is responsible for booking flights provided by a service  $F$  and hotel rooms

available at service  $H$ . The expected interaction can be described as follows:  $U$  makes a booking request by sending a message  $req$  to  $B$  (we will just describe the interaction and abstract away from data details such as trip destination, departure dates and duration). Depending on the request,  $B$  may contact service  $F$  (for booking just a flight ticket),  $H$  (for booking rooms) or both. Service  $B$  uses channels  $reqF$  and  $reqH$  to respectively contact  $F$  and  $H$  (for the sake of simplicity, we assume that any request to  $F$  and  $H$  will be granted). Then, the expected behaviour of  $B$  can be described with the following process:

$$B_0 \stackrel{def}{=} req.(\tau.\overline{reqF} + \tau.\overline{reqH} + \tau.\overline{reqH}.\overline{reqF})$$

In this process, the third branch represents  $B$ 's choice to contact first  $H$  and then  $F$ . Nevertheless, the other partners ( $U$ ,  $F$  and  $H$ ) are not affected in any way by this choice, thus they would be equally happy with alternative definitions such as:

$$\begin{aligned} B_1 &\stackrel{def}{=} req.(\tau.\overline{reqF} + \tau.\overline{reqH} + \tau.\overline{reqF}.\overline{reqH}) \\ B_2 &\stackrel{def}{=} req.(\tau.\overline{reqF} + \tau.\overline{reqH} + \tau.\overline{reqH}.\overline{reqF} + \tau.\overline{reqF}.\overline{reqH}) \end{aligned}$$

Unfortunately,  $B_0$ ,  $B_1$  and  $B_2$  are distinguished by the must testing equivalence. It suffices to consider  $o_0 = \overline{req}.(\tau.1 + reqF.(\tau.1 + reqH.0))$  for showing that  $B_0 \not\sqsubseteq_{\text{must}} B_1$  and that  $B_0 \not\sqsubseteq_{\text{must}} B_2$ , and use  $o_1 = \overline{req}.(\tau.1 + reqH.(\tau.1 + reqF.0))$  for proving that  $B_1 \not\sqsubseteq_{\text{must}} B_2$ .  $\square$

This section is devoted to the definition and characterization of a preorder that is coarser than the classical must preorder and relates processes that cannot be distinguished by distributed contexts. We start by introducing the notion of uncoordinated observers.

**Definition 3.2** (Uncoordinated observer). A process  $\Pi_{i \in 0..n} o_i = o_0 \parallel \dots \parallel o_n$  is an *uncoordinated observer* if  $\mathbf{n}(o_i) \cap \mathbf{n}(o_j) = \emptyset$  for all  $i \neq j$ .  $\square$

Obviously, the condition  $\mathbf{n}(o_i) \cap \mathbf{n}(o_j) = \emptyset$  forbids the direct communication between the sequential components of an uncoordinated observer. As a consequence, a distributed observer cannot impose a total order between actions that are controlled by different components of the observer. Indeed, the executions of a distributed observer are the interleavings of the executions of all sequential components  $\{o_i\}_{i \in 0..n}$  (this property is formally stated in Section 3.1, Lemma 3.5). We remark that a configuration does report success (i.e., perform action  $\checkmark$ ) only when all sequential processes in the composition do report success; an uncoordinated observer reports success when all its components report success simultaneously.

The uncoordinated must testing preorder is obtained by restricting the set of observers to consider just uncoordinated observers over a suitable partition of the interface of a process. We will say  $\mathbb{I} = \{I_i\}_{i \in 0..n}$  is an interface whenever  $\mathbb{I}$  is a partition of  $\text{Act}$  and  $\forall \alpha \in \text{Act}, \alpha \in I_i$  implies  $\overline{\alpha} \in I_i$ . In the remaining of this paper we usually write only the relevant part of an interface. For instance, we will write  $\{\{a\}, \{b\}\}$  for any interface  $\{I_0, I_1\}$  such that  $a \in I_0$  and  $b \in I_1$ .

**Definition 3.3** (Uncoordinated must preorder  $\sqsubseteq_{\text{unc}}^{\mathbb{I}}$ ). Let  $\mathbb{I} = \{I_i\}_{i \in 0..n}$  be an interface. We say  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}} q$  iff for all  $\Pi_{i \in 0..n} o_i$  such that  $\mathbf{n}(o_i) \subseteq I_i$ ,  $p \text{ must } \Pi_{i \in 0..n} o_i$  implies  $q \text{ must } \Pi_{i \in 0..n} o_i$ . We write  $p \approx_{\text{unc}}^{\mathbb{I}} q$  when both  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}} q$  and  $q \sqsubseteq_{\text{unc}}^{\mathbb{I}} p$ .  $\square$

**Example 3.4.** Consider the scenario presented in Example 3.1 and the following interface  $\mathbb{I} = \{\{req\}, \{reqF\}, \{reqH\}\}$  for the process  $B$  that thus interacts with each of the other partners by using a dedicated part of its interface. It can be shown that the three definitions for  $B$  in Example 3.1 are equivalent when considering the uncoordinated must

testing preorder, i.e.,  $B_0 \approx_{\text{unc}}^{\mathbb{I}} B_1 \approx_{\text{unc}}^{\mathbb{I}} B_2$ . The actual proof, which uses the (trace-based) alternative characterization of the preorder, is deferred to Example 3.12.  $\square$

**3.1. Semantic characterisation.** We now address the problem of characterising the uncoordinated must testing preorder in terms of traces and must-sets. In order to do that, we shift from strings to Mazurkiewicz traces [Maz86]. A *Mazurkiewicz trace* is a set of strings, obtained by permuting independent symbols. Traces represent concurrent computations, in which commuting symbols stand for actions that execute independently of one another and non-commuting symbols are causally dependent actions. We start by summarising the basics of the theory of traces in [Maz86].

Let  $D \subseteq \text{Act} \times \text{Act}$  be a finite equivalence relation, called the *dependency relation*, that relates the actions that cannot be commuted. Thus if  $(\alpha, \beta) \in D$ , the two actions have to be considered causally dependent. Symmetrically,  $I_D = (\text{Act} \times \text{Act}) \setminus D$  stands for the *independency* relation with  $(\alpha, \beta) \in I_D$  meaning that  $\alpha$  and  $\beta$  are concurrent.

The trace equivalence induced by the dependency relation  $D$  is the least congruence  $\equiv_D$  in  $\text{Act}^*$  such that for all  $\alpha, \beta \in \text{Act}^*$ :  $(\alpha, \beta) \in I_D \implies \alpha\beta \equiv_D \beta\alpha$ .

The equivalence classes of  $\equiv_D$ , denoted by  $[s]_D$ , are the (Mazurkiewicz) *traces*, namely the strings quotiented via  $\equiv_D$ . The trace monoid, denoted as  $\mathbb{M}(D)$ , is the quotient monoid  $\mathbb{M}(D) = \text{Act}^* / \equiv_D$  whose elements are the traces induced by  $D$ . We remark that no action can commute with  $\checkmark$  because  $I_D$  is defined over  $\text{Act} \times \text{Act}$ .

Let  $\mathbb{I}$  be an interface, the *dependency relation induced by  $\mathbb{I}$*  is  $D = \bigcup_{I \in \mathbb{I}} I \times I$ .

The alternative characterisation of the uncoordinated preorder is defined in terms of equivalence classes of traces. Hence, we extend the transition relation and the notions of convergence and residuals to equivalence classes of strings, as follows:

- $q \xrightarrow{[s]_D} q'$  if and only if  $\exists s' \in [s]_D$  such that  $q \xrightarrow{s'} q'$
- $p \Downarrow [s]_D$  if  $\forall s' \in [s]_D$  then  $p \Downarrow s'$
- $(p \text{ after } [s]_D) = \{p' \mid p \xrightarrow{[s]_D} p'\}$

Now we are able to characterise the behaviour of an uncoordinated observer. We start by formally stating that an uncoordinated observer reaches the same processes after executing any string in an equivalence class. This result is instrumental to the proof of the alternative characterisation of the uncoordinated preorder.

**Lemma 3.5.** *Let  $o = \prod_{i \in 0..n} o_i$  be an uncoordinated observer for the interface  $\mathbb{I} = \{I_i\}_{i \in 0..n}$  and  $D$  the dependency relation induced by  $\mathbb{I}$ . Then, for all  $s \in \text{Act}^*$  and  $t \in [s]_D$  we have  $o \xrightarrow{s} o' \text{ iff } o \xrightarrow{t} o'$ .*

*Proof.* The proof follows by induction on the length  $|s| = |t| = n$ .

- **$n = 0, 1$ .** Immediate, because  $s = t$ .
- **$n > 1$ .** By the Levi's Lemma for traces [Maz95], any possible choice of  $v, w, x, y$  such that  $s = vw$  and  $t = xy$ , implies that  $v \equiv_D z_1 z_2$ ,  $w \equiv_D z_3 z_4$ ,  $x \equiv_D z_1 z_3$ ,  $y \equiv_D z_2 z_4$  with  $(z_2, z_3) \in I_D$ . Consider a decomposition such that  $|v|, |w|, |x|, |y| > 0$  (this is always possible, because  $n > 1$ ). By inductive hypothesis on the reductions  $o \xrightarrow{v} o''$  and  $o'' \xrightarrow{w} o'$ , we have  $o \xrightarrow{v} o'' \xrightarrow{w} o' \text{ iff } o \xrightarrow{z_1} o_1 \xrightarrow{z_2} o_2 \xrightarrow{z_3} o_3 \xrightarrow{z_4} o'$ . Since  $z_2$  and  $z_3$  are independent they take part on different components of the test and  $o_1 \xrightarrow{z_2} o_2 \xrightarrow{z_3} o_3 \text{ iff } o_1 \xrightarrow{z_3} o'_2 \xrightarrow{z_2} o_3$ . Consequently,  $o \xrightarrow{v} o'' \xrightarrow{w} o' \text{ iff } o \xrightarrow{z_1} o_1 \xrightarrow{z_2} o_2 \xrightarrow{z_3} o_3 \xrightarrow{z_4} o' \text{ iff } o \xrightarrow{z_1} o_1 \xrightarrow{z_3} o'_2 \xrightarrow{z_2} o_3 \xrightarrow{z_4} o'$ . By applying inductive hypothesis on reductions  $o \xrightarrow{z_1 z_3} o'_2$  and  $o'_2 \xrightarrow{z_2 z_4} o'$  we have  $o \xrightarrow{z_1} o_1 \xrightarrow{z_2} o'_2 \xrightarrow{z_3} o_3 \xrightarrow{z_4} o' \text{ iff } o \xrightarrow{x} o'' \xrightarrow{y} o'$ .

□

**Lemma 3.6.** *Let  $o = \Pi_{i \in 0..n} o_i$  be an uncoordinated observer for the interface  $\mathbb{I} = \{I_i\}_{i \in 0..n}$  and  $D$  the dependency relation induced by  $\mathbb{I}$ . Then,  $\forall s \in \text{Act}^*, t \in [s]_D$ ,*

- (1)  $s \in \text{str}(o)$  implies  $t \in \text{str}(o)$ .
- (2)  $o \Downarrow s$  implies  $o \Downarrow t$ .
- (3)  $(o \text{ after } s) \text{ MUST } L$  implies  $(o \text{ after } t) \text{ MUST } L$ .
- (4) *If there exists an unsuccessful computation  $o \xRightarrow{s}$ , then there exists an unsuccessful computation  $o \xRightarrow{t}$ .*

The alternative characterisation for the uncoordinated preorder mimics the definition of the classical one, but relies on traces. In the definition below, the condition  $L \subseteq I$  with  $I \in \mathbb{I}$ , captures the idea that each observation is relative to a specific part of the interface.

**Definition 3.7.** Let  $\mathbb{I}$  be an interface and  $D$  the dependency relation induced by  $\mathbb{I}$ . Then,  $p \preceq_{\text{unc}}^{\mathbb{I}} q$  if for every  $s \in \text{Act}^*$ , for any part  $I \in \mathbb{I}$ , for all finite  $L \subseteq I$ , if  $p \Downarrow [s]_D$  then

- (1)  $q \Downarrow [s]_D$
- (2)  $(p \text{ after } [s]_D) \text{ MUST } L$  implies  $(q \text{ after } [s]_D) \text{ MUST } L$

□

The following three lemmata are instrumental to the proof of the correspondence theorem and characterise the relation between the Mazurkiewicz traces of related processes.

**Lemma 3.8.** *Let  $\mathbb{I}$  be an interface and  $D$  the dependency relation induced by  $\mathbb{I}$ . If  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}} q$  then for all  $s \in \text{Act}^*$  we have that  $p \Downarrow [s]_D$  implies*

- (1)  $q \Downarrow [s]_D$
- (2)  $s \in \text{str}(q)$  implies that there exists  $s' \in [s]_D$  such that  $s' \in \text{str}(p)$ .

*Proof.*

- (1) By contradiction. Suppose there exists  $s = a_1 \dots a_n$  such that  $p \Downarrow [s]_D$  and  $q \not\Downarrow [s]_D$ . Then, take the observer  $o = \Pi_{i \in 0..n} o_i$  with  $o_i$  defined as follows

$$o_i = \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k.1) \dots)$$

with  $s \upharpoonright I_i = b_1 \dots b_k$ , we have that  $p \text{ must } o$  and  $q \text{ must } o$ .

- (2) By contradiction. Suppose there exists  $s = a_1 \dots a_n$  such that  $p \Downarrow [s]_D$ ,  $s \in \text{str}(q)$  and  $t \notin \text{str}(p)$  for all  $t \in [s]_D$ . Then, choose the observer  $o = \Pi_{i \in 0..n} o_i$  with  $o_i$  defined as follows

$$\begin{aligned} o_i &= \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k.1) \dots) & \text{if } b_k \neq a_n \\ o_i &= \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k.0) \dots) & \text{if } b_k = a_n \end{aligned}$$

with  $s \upharpoonright I_i = b_1 \dots b_k$ , we have that  $p \text{ must } o$  and  $q \text{ must } o$ .

□

**Lemma 3.9.** *If  $(p \text{ after } [s]_D) \text{ MUST } L$  for some  $L \subseteq \text{Act}$ , then  $\exists t \in [s]_D$  such that  $t \in \text{str}(p)$ .*

*Proof.* Suppose  $\forall t \in [s]_D, t \notin \text{str}(p)$ . Then  $(p \text{ after } [s]_D) = \emptyset$  and, by definition,  $\emptyset \text{ MUST } L$  for every finite  $L \subseteq \text{Act}$ . □

**Lemma 3.10.** *If  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ ,  $s \in \text{str}(q)$  and  $p \Downarrow [s]_D$  then  $\exists s' \in [s]_D$  such that  $s' \in \text{str}(p)$ .*

*Proof.* Assume that  $\forall t \in [s]_D, t \notin \text{str}(p)$ . Then,  $(p \text{ after } [s]_D) = \emptyset$ . By Lemma 3.9,  $(p \text{ after } [s]_D) \text{ MUST } L$  for every finite  $L \subseteq A$ . Since  $p \Downarrow [s]_D$  and  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ ,  $q \Downarrow [s]_D$ . Hence, the set  $\bigcup \{\text{init}(q') \mid q \xRightarrow{[s]_D} q'\}$  is finite. Therefore, we can find an action  $a$  such that  $q \not\xRightarrow{a}$ . Then  $(q \text{ after } [s]_D) \text{ MUST } \{a\}$  while  $(p \text{ after } [s]_D) \text{ MUST } \{a\}$ , which contradicts the hypothesis  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ . □



**Theorem 3.11.**  $\sqsubseteq_{\text{unc}}^{\mathbb{I}} = \preceq_{\text{unc}}^{\mathbb{I}}$ .

*Proof.*

⊆) Actually we prove that  $p \not\sqsubseteq_{\text{unc}}^{\mathbb{I}} q$  implies  $p \not\sqsubseteq_{\text{unc}}^{\mathbb{I}} q$ . Let  $D$  be the dependency relation induced by  $\mathbb{I}$ . Assume that there exists  $s = a_1 \dots a_n$  and  $I_j \in \mathbb{I}$  and  $L \subseteq I_j$  such that

- (1)  $p \Downarrow [s]_D$  and  $q \Uparrow [s]_D$ , or
- (2)  $s \in \text{str}(q)$  and  $\forall t \in [s]_D. t \notin \text{str}(p)$  or
- (3)  $(p \text{ after } [s]_D) \text{ MUST } L$  and  $(q \text{ after } [s]_D) \text{ M\Upsilon ST } L$

For each case we show that there exists an observer such that  $p \text{ must } o$  and  $q \text{ m\Upsilon st } o$ . For the two first cases, we take the observers as defined in proof of Lemma 3.8. For the third one, we take  $o = \prod_{i \in 0, \dots, n} o_i$  with  $o_i$  defined as follows

$$\begin{aligned} o_i &= \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k.1) \dots) & \text{if } i \neq j \\ o_i &= \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k. \sum_{a \in L} a.1) \dots) & \text{if } i = j \end{aligned}$$

with  $s \upharpoonright I_j = b_1 \dots b_k$ .

⊇) We prove  $p \preceq_{\text{unc}}^{\mathbb{I}} q$  implies  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}} q$ . Actually, the proof follows by showing that  $p \preceq_{\text{unc}}^{\mathbb{I}} q$  and  $q \text{ m\Upsilon st } o$  imply  $p \text{ must } o$ . Assume there exists an unsuccessful computation

$$q \parallel o = q_0 \parallel o_0 \xrightarrow{\tau} \dots \xrightarrow{\tau} q_k \parallel o_k \xrightarrow{\tau} \dots$$

Consider the following cases:

- (1) **The computation is finite**, i.e., there exists  $n$  such that  $q_n \parallel o_n \not\xrightarrow{\tau}$  and  $q_i \Downarrow$  and  $o_i \Downarrow$  for  $i \leq n$ . By unzipping the computation we have  $q_0 \xrightarrow{s} q_n$  and  $o_0 \xrightarrow{\bar{s}} o_n$ , which is unsuccessful. Moreover,  $q_n \text{ M\Upsilon ST init}(o_n)$  and, hence  $(q \text{ after } [s]_D) \text{ M\Upsilon ST init}(o_n)$ .
  - (a) Case  $p \Uparrow [s]_D$ , i.e.,  $\exists t \in [s]_D$  and  $p \Uparrow t$ . By Corollary 3.6(4),  $o \xrightarrow{\bar{s}} \text{ implies } o \xrightarrow{\bar{t}}$  also unsuccessful, and hence there is an unsuccessful computation of  $p \parallel o$ .
  - (b) Case  $p \Downarrow [s]_D$ . Note that  $s \in \text{str}(q)$ , then  $\exists t \in [s]_D : t \in \text{str}(p)$ , by Lemma 3.8(2). Hence  $(p \text{ after } [s]_D) \neq \emptyset$ . Since  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ , we have that  $(q \text{ after } [s]_D) \text{ M\Upsilon ST init}(o_n)$  implies  $(p \text{ after } [s]_D) \text{ M\Upsilon ST init}(o_n)$ . Therefore, exists some  $p' \in (p \text{ after } [s]_D)$  and  $p' \text{ M\Upsilon ST init}(o_n)$ . Hence,  $\exists t' \in [s]_D. p \xrightarrow{t'}$ . By Corollary 3.6(4),  $o \xrightarrow{\bar{t}}$  unsuccessful, and hence there is an unsuccessful computation of  $p \parallel o$ .
- (2) **The computation is infinite**. We consider two cases:
  - (a) There exists  $s \in \text{str}(q)$  and  $\bar{s} \in \text{str}(o)$  such that  $q \Uparrow s$  or  $o \Uparrow \bar{s}$ . We proceed by case analysis.
    - $q \Uparrow [s]_D$ : Since  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ ,  $p \Uparrow [s]_D$ . Therefore,  $\exists t \in [s]_D$  such that  $p \Uparrow t$ . By Corollary 3.6(4),  $o \xrightarrow{\bar{t}}$  unsuccessful, and hence there is an unsuccessful computation of  $p \parallel o$ .
    - $q \Downarrow [s]_D$  (and  $o \Uparrow \bar{s}$ ): By Lemma 3.10,  $\exists t \in [s]_D : t \in \text{str}(p)$ . By Corollary 3.6(2),  $o \Uparrow \bar{t}$ , and hence there is an unsuccessful computation of  $p \parallel o$ .
  - (b)  $\forall n. q_n \Downarrow$  and  $o_n \Downarrow$ . Take  $s \in \text{Act}^*$  such that  $q \xrightarrow{s} q_n$  and  $q \Downarrow s$  and reason analogously to case 1 (i.e., by considering either  $p \Uparrow [s]_D$  or  $p \Downarrow [s]_D$ ) to prove that there exists an unsuccessful computation of  $p \parallel o$ .

□

In the following we will write  $L_{p, [s]_D}^I$  for the smallest set such that  $(p \text{ after } [s]_D) \text{ MUST } L$  and  $L \subseteq I$  imply  $L_{p, [s]_D}^I \subseteq L$ .

**Example 3.12.** We take advantage of the alternative characterisation of the uncoordinated preorder to show that the three processes for the broker in Example 3.1 are equivalent when

considering  $\mathbb{I} = \{\{req\}, \{reqF\}, \{reqH\}\}$ . Actually, we will only consider  $B_0 \approx_{\text{unc}}^{\mathbb{I}} B_1$ , being that the proofs for  $B_0 \approx_{\text{unc}}^{\mathbb{I}} B_2$  and  $B_1 \approx_{\text{unc}}^{\mathbb{I}} B_2$  are analogous.

Firstly, we have to consider that  $B_0 \Downarrow s$  and  $B_1 \Downarrow s$  for any  $s$  because  $B_0$  and  $B_1$  do not have infinite computations. The relation between must-sets are described in the two tables below. The first table shows the sets  $(B_0 \text{ after } [s]_D)$  and  $L_{B_0, [s]_D}^I$ . Note that  $[s]_D$  in the first column will be represented by any string  $s' \in [s]_D$ . Moreover, we write “—” in the three last columns whenever  $L_{B_0, [s]_D}^I$  does not exist. The second table does the same for  $B_1$ . In the tables, we let  $B'_0$  stand for  $\tau.\overline{reqF} + \tau.\overline{reqH} + \tau.\overline{reqH}.\overline{reqF}$  and  $B'_1$  stand for  $\tau.\overline{reqF} + \tau.\overline{reqH} + \tau.\overline{reqF}.\overline{reqH}$ .

$[s]_D$	$B_0 \text{ after } [s]_D$	$L_{B_0, [s]_D}^{\{req\}}$	$L_{B_0, [s]_D}^{\{reqH\}}$	$L_{B_0, [s]_D}^{\{reqF\}}$
$\epsilon$	$B_0$	$\{req\}$	—	—
$req$	$\{B'_0, \overline{reqF}, \overline{reqH}, \overline{reqH}.\overline{reqF}\}$	—	—	—
$req.\overline{reqF}$	$\{0\}$	—	—	—
$req.\overline{reqH}$	$\{0, \overline{reqF}\}$	—	—	—
$req.\overline{reqF}.\overline{reqH}$	$\{0\}$	—	—	—
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

  

$[s]_D$	$B_1 \text{ after } [s]_D$	$L_{B_1, [s]_D}^{\{req\}}$	$L_{B_1, [s]_D}^{\{reqH\}}$	$L_{B_1, [s]_D}^{\{reqF\}}$
$\epsilon$	$B_1$	$\{req\}$	—	—
$req$	$\{B'_1, \overline{reqF}, \overline{reqH}, \overline{reqF}.\overline{reqH}\}$	—	—	—
$req.\overline{reqF}$	$\{0, \overline{reqH}\}$	—	—	—
$req.\overline{reqH}$	$\{0\}$	—	—	—
$req.\overline{reqF}.\overline{reqH}$	$\{0\}$	—	—	—
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

By inspecting the tables, we can check that for any possible trace  $[s]_D$  and  $I \in \mathbb{I}$ , it holds that  $L_{B_0, [s]_D}^I = L_{B_1, [s]_D}^I$ . Consequently,  $(B_0 \text{ after } [s]_D) \text{ MUST } L$  iff  $(B_1 \text{ after } [s]_D) \text{ MUST } L$  and thus we have  $B_0 \approx_{\text{unc}}^{\mathbb{I}} B_1$ .

We now present two additional examples that help us in understanding the discriminating capability of the uncoordinated preorder, its differences with the classical must preorder and its adequacy for modelling process conformance.

The first of these examples shows that a process that does not communicate its internal choices to all of its clients is useless in a distributed context.

**Example 3.13.** Consider the process  $p = \tau.a + \tau.b$  that is intended to be used by two partners with the following interface:  $\mathbb{I} = \{\{a\}, \{b\}\}$ . We show that this process is less useful than 0 in an uncoordinated context, i.e.,  $\tau.a + \tau.b \sqsubseteq_{\text{unc}}^{\mathbb{I}} 0$ . It is immediate to see that  $p$  and 0 strongly converge for any  $s \in \text{Act}^*$ , then the minimal sets  $L_{p, [s]_D}^{\{a\}}$ ,  $L_{p, [s]_D}^{\{b\}}$ ,  $L_{0, [s]_D}^{\{a\}}$  and  $L_{0, [s]_D}^{\{b\}}$  presented in the tables below are sufficient for proving our claim.

$[s]_D$	$p \text{ after } [s]_D$	$L_{p, [s]_D}^{\{a\}}$	$L_{p, [s]_D}^{\{b\}}$
$\epsilon$	$p, a, b$	—	—
$a$	$\{0\}$	—	—
$b$	$\{0\}$	—	—
other	$\emptyset$	$\emptyset$	$\emptyset$

  

$[s]_D$	$0 \text{ after } [s]_D$	$L_{0, [s]_D}^{\{a\}}$	$L_{0, [s]_D}^{\{b\}}$
$\epsilon$	0	—	—
$a$	$\emptyset$	$\emptyset$	$\emptyset$
$b$	$\emptyset$	$\emptyset$	$\emptyset$
other	$\emptyset$	$\emptyset$	$\emptyset$

Note that differently from the classical must preorder, the uncoordinated preorder does not consider the must-set  $\{a, b\}$  to distinguish  $p$  from 0 because this set involves channels

in different parts of the interface. The key point here is that each internal reduction of  $p$  is observed just by one part of the interface: the choice of branch  $a$  is only observed by one client and the choice of  $b$  is observed by the other one. Since uncoordinated observers do not intercommunicate, they can only report success simultaneously if they can do it independently from the interactions with the tested process, but such observers are exactly the ones that 0 can pass.

Like in the classical must preorder, we have that  $0 \not\sqsubseteq_{\text{unc}}^{\mathbb{I}} \tau.a + \tau.b$ . This is witnessed by the observer  $o = \bar{a}.0 + \tau.1 \parallel 1$  that is passed by 0 but not by  $\tau.a + \tau.b$ .  $\square$

The second example shows that the uncoordinated preorder falls somehow short with respect to the target we set in the introduction of allowing servers to swap actions that are targeted to different clients.

**Example 3.14.** Consider the interface  $\mathbb{I} = \{\{a\}, \{b\}\}$  and the two pairs of processes

- $a.b + a + b$  and  $b.a + a + b$
- $a.b$  and  $b.a$

By inspecting traces and must-sets in the two tables below, where we use  $p$  and  $q$  to denote  $a.b + a + b$  and  $b.a + a + b$

$[s]_D$	$p$ after $[s]_D$	$L_{p,[s]_D}^{\{a\}}$	$L_{p,[s]_D}^{\{b\}}$
$\epsilon$	$\{p\}$	$\{a\}$	$\{b\}$
$a$	$\{b, 0\}$	—	—
$b$	$\{0\}$	—	—
$ab$	$\{0\}$	—	—
other	$\emptyset$	$\emptyset$	$\emptyset$

$[s]_D$	$q$ after $[s]_D$	$L_{q,[s]_D}^{\{a\}}$	$L_{q,[s]_D}^{\{b\}}$
$\epsilon$	$\{p\}$	$\{a\}$	$\{b\}$
$a$	$\{0\}$	—	—
$b$	$\{a, 0\}$	—	—
$ab$	$\{0\}$	—	—
other	$\emptyset$	$\emptyset$	$\emptyset$

It is easy to see that

$$a.b + a + b \approx_{\text{unc}}^{\mathbb{I}} b.a + a + b$$

However, by using  $o = \bar{a}.1 \parallel 1$  and  $o' = 1 \parallel \bar{b}.1$  as observers, it can be shown that

$$a.b \not\sqsubseteq_{\text{unc}}^{\mathbb{I}} b.a \quad \text{and} \quad b.a \not\sqsubseteq_{\text{unc}}^{\mathbb{I}} a.b$$

Note that  $o = \bar{a}.1 \parallel 1$  actually interacts with the process under test by using just one part of the interface and relies on the fact that the remaining part of the interface stays idle. Thanks to this ability, uncoordinated observers have still a limited power to track some dependencies among actions on different parts of the interface.

The preorder presented in the next section limits further the discriminating power of observers and allows us to equate processes  $a.b$  and  $b.a$ .  $\square$

#### 4. A TESTING PREORDER WITH INDIVIDUALISTIC OBSERVERS

In this section we explore a notion of equivalence equating processes that can freely permute actions over different parts of their interface. As for the uncoordinated observers, the targeted scenario is that of a service with a partitioned interface interacting with two or more independent partners by using separate sets of ports. In addition, each component of an observer cannot exploit any knowledge about the design choices made by the other components, i.e., each of them has a local view of the behaviour of the process that ignores all actions controlled by the remaining components. Local views are characterised in terms of a projection operator defined as follows.

**Definition 4.1** (Projection). Let  $V \subseteq \mathcal{N}$  be a set of observable ports. We write  $p \upharpoonright V$  for the process obtained by hiding all actions of  $p$  over channels that are not in  $V$ . Formally,

$$\frac{p \xrightarrow{\alpha} p' \quad \alpha \in V \cup \overline{V}}{p \upharpoonright V \xrightarrow{\alpha} p' \upharpoonright V} \qquad \frac{p \xrightarrow{\alpha} p' \quad \alpha \notin V \cup \overline{V}}{p \upharpoonright V \xrightarrow{\tau} p' \upharpoonright V}$$

□

**Definition 4.2** (Individualistic (must) preorder  $\sqsubseteq_{\text{ind}}^{\mathbb{I}}$ ). Let  $\mathbb{I} = \{I_i\}_{i \in 0..n}$  be an interface. We say  $p \sqsubseteq_{\text{ind}}^{\mathbb{I}} q$  iff for all  $\Pi_{i \in 0..n} o_i$  such that  $\mathbf{n}(o_i) \subseteq I_i$ ,  $p \upharpoonright I_i \text{ must } o_i$  implies  $q \upharpoonright I_i \text{ must } o_i$ . □

Note that  $a.b$  and  $b.a$  cannot be distinguished anymore by the observer  $o = \overline{a}.1 \parallel \mathbf{1}$  used in the previous section to prove  $a.b \not\sqsubseteq_{\text{unc}}^{\{\{a\}, \{b\}\}} b.a$  (Example 3.14), because  $a.b \upharpoonright \{a\} \text{ must } \overline{a}.1$ ,  $b.a \upharpoonright \{a\} \text{ must } \overline{a}.1$ ,  $a.b \upharpoonright \{b\} \text{ must } \mathbf{1}$  and  $b.a \upharpoonright \{b\} \text{ must } \mathbf{1}$ . Indeed, later (Example 4.10) we will see that:

$$a.b \approx_{\text{ind}}^{\{\{a\}, \{b\}\}} b.a$$

**4.1. Semantic characterisation.** In this section we address the characterisation of the individualistic preorder in terms of traces. We start by introducing an equivalence notion of traces that ignores hidden actions.

**Definition 4.3** (Filtered traces). Let  $I \subseteq \text{Act}$ . Two strings  $s, t \in \text{Act}^*$  are *equivalent up-to*  $I$ , written  $s \equiv_I$ , if and only if  $s \upharpoonright I = t \upharpoonright I$ . We write  $[[s]]_I$  for the equivalence class of  $s$ . □

Basically, two traces are equivalent up-to  $I$  when they coincide after the removal of hidden actions. As for the distributed preorder, we extend the notions of reduction, convergence and residuals to equivalence classes of filtered traces.

- $q \xrightarrow{[[s]]_I} q'$  if and only if  $\exists t \in [[s]]_I$  such that  $q \xrightarrow{t} q'$
- $p \Downarrow [[s]]_I$  if and only if  $\forall t \in [[s]]_I. p \Downarrow t$
- $(p \text{ after } [[s]]_I) = \{p' \mid p \xrightarrow{[[s]]_I} p'\}$

The following auxiliary result establishes properties relating reductions, hiding and filtered traces, which will be useful in the proof of the correspondence theorem.

**Lemma 4.4.**

- (1)  $p \xrightarrow{s} p'$  implies  $p \upharpoonright I \xrightarrow{s \upharpoonright I} p' \upharpoonright I$ .
- (2)  $p \upharpoonright I \xrightarrow{s} p' \upharpoonright I$  implies  $\exists t \in [[s]]_I$  and  $p \xrightarrow{t} p'$ .
- (3)  $p \upharpoonright [[s]]_I$  implies  $p \upharpoonright I \upharpoonright s \upharpoonright I$ .
- (4)  $(p \text{ after } [[s]]_I) \text{ MUST } L$  iff  $(p \upharpoonright I \text{ after } s \upharpoonright I) \text{ MUST } L \cap I$ .

*Proof.* The proof follows by induction on the length of  $s$ . □

The alternative characterisation for the individualistic preorder is given in terms of filtered traces.

**Definition 4.5.** Let  $p \preceq_{\text{ind}}^{\mathbb{I}} q$  if for every  $I \in \mathbb{I}$ , for every  $s \in I^*$ , and for all finite  $L \subseteq I$ , if  $p \Downarrow [[s]]_I$  then

- (1)  $q \Downarrow [[s]]_I$
- (2)  $(p \text{ after } [[s]]_I) \text{ MUST } L \cup (\text{Act} \setminus I)$  implies  $(q \text{ after } [[s]]_I) \text{ MUST } L \cup (\text{Act} \setminus I)$

We would like to draw attention to condition 2 above; it only considers must-sets that always include all the actions in  $(\text{Act} \setminus I)$  to avoid the possibility of distinguishing reachable states because of actions that are not in  $I$ . Consider that this condition could be formulated as follows: for all finite  $L \subseteq \text{Act}$ ,

$(p \text{ after } [[s]]_I) \text{ MUST } L$  implies  $\exists L'$  such that  $(q \text{ after } [[s]]_I) \text{ MUST } L'$  and  $L \cap I = L' \cap I$

which makes evident that only the actions from the observable part of the interface are relevant.

The following two lemmata are analogous to those for the uncoordinated preorder.

**Lemma 4.6.** *If  $p \sqsubseteq_{\text{ind}}^{\mathbb{I}} q$  then for all  $s \in \text{Act}^*$  and  $I \in \mathbb{I}$ , we have that  $p \Downarrow [[s]]_I$  implies*

- (1)  $q \Downarrow [[s]]_I$
- (2)  $s \in \text{str}(q)$  implies that there exists  $t \in [[s]]_I$  such that  $t \in \text{str}(p)$ .

*Proof.*

- (1) By contradiction. Suppose there exists  $s = a_1 \dots a_n$  such that  $p \Downarrow [[s]]_I$  and  $q \Uparrow [[s]]_I$ . Then, take the observer  $o = \Pi_{i \in 0, \dots, n} o_i$  with  $o_i$  defined as follows

$$o_i = \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k.1) \dots)$$

with  $s \upharpoonright I = b_1 \dots b_k$ . Then,  $p \upharpoonright I \text{ must } o_i$  and  $q \upharpoonright I \text{ must } o_i$ .

- (2) By contradiction. Suppose there exists  $s = a_1 \dots a_n$  such that  $p \Downarrow [[s]]_I$ ,  $s \in \text{str}(q)$  and for all  $t \in [[s]]_I$ ,  $t \notin \text{str}(p)$ . Then, choose  $o = \Pi_{i \in 0, \dots, n} o_i$  with  $o_i$  defined as follows

$$o_i = \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k.0) \dots)$$

with  $s \upharpoonright I = b_1 \dots b_k$ . Then,  $p \upharpoonright I \text{ must } o_i$  and  $q \upharpoonright I \text{ must } o_i$ . □

**Lemma 4.7.** *if  $(p \text{ after } [[s]]_I) \text{ MUST } L$  for some  $L \subseteq \text{Act}$ , then  $\exists t \in [[s]]_I : t \in \text{str}(p)$ .*

*Proof.* Suppose  $\forall t \in [[s]]_I$ ,  $t \notin \text{str}(p)$ . Then  $(p \text{ after } [[s]]_I) = \emptyset$  and, by definition,  $\emptyset \text{ MUST } L$  for every finite  $L \subseteq \text{Act}$ . □

We rely on the following auxiliary results relating the traces of processes in the must preorders.

**Lemma 4.8.** *If  $p \preceq_{\text{ind}}^{\mathbb{I}} q$ ,  $s \in \text{str}(q)$  and  $p \Downarrow [[s]]_I$  with  $I \in \mathbb{I}$  then  $t \in ([s]]_I \cap \text{str}(p))$ .*

*Proof.* Assume  $\forall t \in [[s]]_I : t \notin \text{str}(p)$ . By Lemma 4.7,  $(p \text{ after } [[s]]_I) \text{ MUST } L$  for every finite  $L \subseteq \text{Act}$ . Since  $p \Downarrow [[s]]_I$ , the set  $\bigcup \{\text{init}(q') \mid q \xrightarrow{[[s]]_I} q'\}$  is finite. Therefore, we can find an action  $a$  such that for all  $t \in [[s]]_I$  we have  $q \not\xrightarrow{t a}$ . Then  $(q \text{ after } [[s]]_I) \text{ MUST } \{a\}$  while  $(p \text{ after } [[s]]_I) \text{ MUST } \{a\}$ , which contradicts the hypothesis  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ . □

**Theorem 4.9.**  $\sqsubseteq_{\text{ind}}^{\mathbb{I}} = \preceq_{\text{ind}}^{\mathbb{I}}$ .

*Proof.*

- ⊆) Actually we prove that  $p \not\preceq_{\text{ind}}^{\mathbb{I}} q$  implies  $p \not\sqsubseteq_{\text{ind}}^{\mathbb{I}} q$ . Assume that there exists  $s = a_1 \dots a_n$  and  $L \subseteq I$  such that

- (1)  $p \Downarrow [[s]]_I$  and  $q \Uparrow [[s]]_I$ , or
- (2)  $s \in \text{str}(q)$  and  $\forall t \in [[s]]_I. t \notin \text{str}(p)$  or
- (3)  $(p \text{ after } [[s]]_I) \text{ MUST } L \cup (\text{Act} \setminus I)$  and  $(q \text{ after } [[s]]_I) \text{ MUST } L \cup (\text{Act} \setminus I)$

For each case we show that there exists an observer such that  $p \upharpoonright I \text{ must } o$  and  $q \upharpoonright I \text{ must } o$ . For the two first cases, we take the observers as defined in proof of Lemma 4.6. For the third one, define

$$o = \tau.1 + b_1.(\tau.1 + \dots (\tau.1 + b_k. \sum_{a \in L} a.1) \dots)$$

with  $s \upharpoonright I = b_1 \dots b_k$

$\supseteq$ ) We prove  $p \preceq_{\text{ind}}^I q$  implies  $p \sqsubseteq_{\text{ind}}^I q$ . Actually, the proof follows by showing that  $p \preceq_{\text{ind}}^I q$  and  $q \upharpoonright I \text{ must } o$  imply  $p \upharpoonright I \text{ must } o$ . Assume there exists an unsuccessful computation

$$q \upharpoonright I \parallel o = q_0 \upharpoonright I \parallel o_0 \xrightarrow{\tau} \dots \xrightarrow{\tau} q_k \upharpoonright I \parallel o_k \xrightarrow{\tau} \dots$$

Consider the following cases:

- (1) **The computation is finite**, i.e., there exists  $n$  such that  $q_n \upharpoonright I \parallel o_n \not\rightarrow$  and  $q_i \upharpoonright I \Downarrow$  and  $o_i \Downarrow$  for  $i \leq n$ . By unzipping the computation, there exists  $s$  such that  $q_0 \upharpoonright I \xrightarrow{s} q_n \upharpoonright I$  and  $o_0 \xrightarrow{\bar{s}} o_n$  unsuccessful. Note that  $\mathbf{n}(s) \subseteq I$  and hence  $s \upharpoonright I = s$ . Moreover,  $q_n \upharpoonright I \text{ MUST init}(o_n)$  and, hence  $(q \upharpoonright I \text{ after } s) \text{ MUST init}(o_n)$ . By Lemma 4.4(4), we have that  $(q \text{ after } [[s]]_I) \text{ MUST init}(o_n)$ .
  - (a) Case  $p \upharpoonright I \uparrow [[s]]_I$ . By Lemma 4.4(3),  $p \upharpoonright I \uparrow s$ . Consequently, there is an unsuccessful computation of  $p \upharpoonright I \parallel o$ .
  - (b) Case  $p \Downarrow [[s]]_I$ . Note that  $s \in \text{str}(q \upharpoonright I)$ . By Lemma 4.6(2), therefore,  $\exists t \in [[s]]_I$  and  $t \in \text{str}(p)$ . Hence,  $(p \text{ after } [[s]]_I) \neq \emptyset$ . Moreover, from  $p \preceq_{\text{ind}}^I q$  we conclude that  $(q \text{ after } [[s]]_I) \text{ MUST init}(o)$  implies  $(p \text{ after } [[s]]_I) \text{ MUST init}(o_n)$ . Therefore, exists some  $p' \in (p \text{ after } [[s]]_I)$  such that  $p' \text{ MUST init}(o_n)$ , and  $p \upharpoonright I \xrightarrow{s} p' \upharpoonright I$  is unsuccessful. Hence, there is an unsuccessful computation of  $p \upharpoonright I \parallel o$ .
- (2) **The computation is infinite**. We consider two cases:
  - (a) There exists  $s \in \text{str}(q \upharpoonright I)$  and  $\bar{s} \in \text{str}(o)$  such that  $q \upharpoonright I \uparrow s$  or  $o \uparrow \bar{s}$ . Note that  $\mathbf{n}(s) \subseteq I$  and hence  $s \upharpoonright I = s$ . We proceed by case analysis.
    - $q \upharpoonright I \uparrow s$ : By Lemma 4.4(3),  $q \upharpoonright I \uparrow [[s]]_I$ . Therefore  $p \upharpoonright I \uparrow [[s]]_I$  because  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ . By Lemma 4.4(3),  $p \upharpoonright I \uparrow s$ . Therefore, there is an unsuccessful computation of  $p \upharpoonright I \parallel o$ .
    - $q \upharpoonright I \Downarrow s$  (and  $o \uparrow \bar{s}$ ): Therefore  $q \Downarrow [[s]]_I$  by Lemma 4.4(3). Therefore  $p \Downarrow [[s]]_I$  because  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ . By Lemma 4.8,  $\exists t \in [[s]]_I : t \in \text{str}(p)$ , hence  $p \xrightarrow{t} p'$ . By Lemma 4.4(1)  $p \upharpoonright I \xrightarrow{t|I} p' \upharpoonright I$ . Note that  $t \upharpoonright I = s$ . Then,  $p \upharpoonright I \xrightarrow{s} p' \upharpoonright I$ . Hence, the computation obtained by zipping  $p \upharpoonright I \xrightarrow{s} p' \upharpoonright I$  and  $o \xrightarrow{\bar{s}} o'$  is unsuccessful.
  - (b)  $\forall n. q_n \Downarrow$  and  $o_n \Downarrow$ . Take  $s \in \text{Act}^*$  such that  $q \xrightarrow{s} q_n$  and  $q \Downarrow s$  and reason analogously to case 1 (i.e., by considering either  $p \upharpoonright I \uparrow [s]_D$  or  $p \Downarrow [s]_D$ ) to prove that there exists an unsuccessful computation of  $p \parallel o$ .

□

**Example 4.10.** Consider the processes  $p = a.b$  and  $q = b.a$  and the interface  $\mathbb{I} = \{\{a\}, \{b\}\}$ . The table below shows the analysis for the part of the interface  $\{a\}$ .

$[[s]]_{\{a\}}$	$p \text{ after } [[s]]_{\{a\}}$	$L_{p, [[s]]_I}^{\{a\}}$	$q \text{ after } [[s]]_{\{a\}}$	$L_{q, [[s]]_I}^{\{a\}}$
$\epsilon$	$\{p\}$	$\{a\}$	$\{q, a\}$	$\{a\}$
$a$	$\{0, b\}$	—	$\{0\}$	—
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

When analysing the sets  $(p \text{ after } [[\epsilon]]_{\{a\}}) = \{p\}$  and  $(q \text{ after } [[\epsilon]]_{\{a\}}) = \{q, a\}$ , we ignore the fact that  $q$  starts with a hidden action  $b$ ; the only relevant residuals are those performing  $a$ . With a similar analysis we conclude that the condition on must-sets also holds for set  $\{b\}$ . Hence,  $a.b \approx_{\text{ind}}^{\mathbb{I}} b.a$  holds.  $\square$

The following example illustrates also the fact that individualistic observers are unable to track causal dependencies between choices made in different parts of the interface.

**Example 4.11.** Let  $p_1 = a.c + b.d$  and  $p_2 = a.d + b.c$  be two alternative implementations for a service with interface  $\mathbb{I} = \{\{a, b\}, \{c, d\}\}$ . These two implementations are distinguished by the uncoordinated preorder ( $p_1 \not\approx_{\text{unc}}^{\{\{a, b\}, \{c, d\}\}} p_2$ ) because of the observers  $o_1 = \bar{a}.1 \parallel \bar{c}.1$  ( $p_1 \not\sqsubseteq_{\text{unc}}^{\{\{a, b\}, \{c, d\}\}} p_2$ ) and  $o_2 = \bar{b}.1 \parallel \bar{c}.1$  ( $p_2 \not\sqsubseteq_{\text{unc}}^{\{\{a, b\}, \{c, d\}\}} p_1$ ).

They are instead equated by the individualistic preorder,  $p_1 \approx_{\text{ind}}^{\mathbb{I}} p_2$ . Indeed, if only the part of the interface  $\{a, b\}$  is of interest, we have that  $p_1$  and  $p_2$  are equivalent because they exhibit the same interactions over channels  $a$  and  $b$ . Similarly, without any a priori knowledge of the choices made for  $\{a, b\}$ , the behaviour observed over  $\{c, d\}$  can be described by the non-deterministic choice  $\tau.c + \tau.d$ , and hence,  $p_1$  and  $p_2$  are indistinguishable also over  $\{c, d\}$ .

We use the alternative characterisation to prove our claim. As usual,  $p_1 \Downarrow s$  and  $p_2 \Downarrow s$  for any  $s$ . The tables below show coincidence of the must-sets. We would only like to remark that  $ac \in [[a]]_{\{a, b\}}$  and, consequently,  $p_1 \text{ after } [[a]]_{\{a, b\}}$  contains also process 0.

$[[s]]_{\{a, b\}}$	$p_1 \text{ after } [[s]]_{\{a, b\}}$	$L_{p_1, [[s]]_I}^{\{a, b\}}$	$p_2 \text{ after } [[s]]_{\{a, b\}}$	$L_{p_2, [[s]]_I}^{\{a, b\}}$
$\epsilon$	$p_1$	$\{a, b\}$	$p_2$	$\{a, b\}$
$a$	$\{c, 0\}$	—	$\{d, 0\}$	—
$b$	$\{d, 0\}$	—	$\{c, 0\}$	—
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

  

$[[s]]_{\{c, d\}}$	$p_1 \text{ after } [[s]]_{\{a, b\}}$	$L_{p_1, [[s]]_I}^{\{c, d\}}$	$p_2 \text{ after } [[s]]_{\{a, b\}}$	$L_{p_2, [[s]]_I}^{\{c, d\}}$
$\epsilon$	$p_1$	$\{c, d\}$	$p_2$	$\{c, d\}$
$c$	$\{0\}$	—	$\{0\}$	—
$d$	$\{0\}$	—	$\{0\}$	—
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$\square$

## 5. RELATION BETWEEN MUST, UNCOORDINATED AND INDIVIDUALISTIC PREORDERS

In this section, we formally study the relationships between the classical must preorder and the two preorders we have introduced. We start by showing that a refinement of an interface induces a coarser preorder, e.g., splitting the observation among more uncoordinated observers decreases the discriminating power of the tests. We say that an interface  $\mathbb{I}'$  is a *refinement* of another interface  $\mathbb{I}$  when the partition  $\mathbb{I}'$  is finer than the partition  $\mathbb{I}$ .

**Lemma 5.1.** *Let  $\mathbb{I}$  be an interface and  $\mathbb{I}'$  a refinement of  $\mathbb{I}$ . Then,  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}} q$  implies  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}'} q$ .*

*Proof.* The proof follows by showing that  $p \preceq_{\text{unc}}^{\mathbb{I}} q$  implies  $p \preceq_{\text{unc}}^{\mathbb{I}'} q$ . Let  $D$  and  $D'$  be the dependency relations induced respectively by  $\mathbb{I}$  and  $\mathbb{I}'$ . Since  $\mathbb{I}'$  is a refinement of  $\mathbb{I}$ ,  $D' \subseteq D$  and therefore  $[s]_D \subseteq [s]_{D'}$  for all  $s$ . Assume  $p \Downarrow [s]_{D'}$  for  $s \in \text{Act}^*$ . Then,

- $p \Downarrow t$  for all  $t \in [s]_{D'}$ . Note that  $[t]_{D'} = [s]_{D'}$  because  $t \in [s]_{D'}$ . Consequently,  $p \Downarrow [t]_D$  because  $[t]_D \subseteq [t]_{D'} = [s]_{D'}$ . Since  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ , we know that  $q \Downarrow [t]_D$ , which implies  $q \Downarrow t$ . Therefore,  $q \Downarrow [s]_{D'}$ .
- Assume  $L \subseteq I'$ ,  $I' \in \mathbb{I}'$  and  $(p \text{ after } [s]_{D'}) \text{ MUST } L$ . Then,  $(p \text{ after } t) \text{ MUST } L$  for all  $t \in [s]_{D'}$ . Therefore,  $(p \text{ after } [t]_D) \text{ MUST } L$  because  $[t]_D \subseteq [t]_{D'} = [s]_{D'}$ . Since  $\mathbb{I}'$  is a refinement of  $\mathbb{I}$ , there is some  $I \in \mathbb{I}$  such that  $I' \subseteq I$  and  $L \subseteq I$  and  $I \in \mathbb{I}$ . Consequently,  $p \preceq_{\text{unc}}^{\mathbb{I}} q$  implies  $(q \text{ after } [t]_D) \text{ MUST } L$  and, hence,  $(q \text{ after } t) \text{ MUST } L$  for all  $t \in [s]_{D'}$ . Therefore,  $(q \text{ after } [s]_{D'}) \text{ MUST } L$ .

□

This result allows us to conclude that the uncoordinated preorder is coarser than the classical must testing preorder. It suffices to note that the preorder associated to the maximal element of the partition lattice, i.e., the trivial partition  $\mathbb{I} = \{\text{Act}\}$ , corresponds to  $\sqsubseteq_{\text{must}}$ .

**Proposition 5.2.** *Let  $\mathbb{I}$  be an interface. Then,  $p \sqsubseteq_{\text{must}} q$  implies  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}} q$ .*

*Proof.* It follows from Lemma 5.1 by noting that the preorder associated to the maximal element in the partition lattice, i.e., trivial partition  $\tilde{\mathbb{I}} = \{\text{Act}\}$ , corresponds to  $\sqsubseteq_{\text{must}}$ . In fact,  $\tilde{\mathbb{I}}$  induces a total dependency relation  $\tilde{D}$ . This implies  $[s]_{\tilde{D}} = \{s\}$  for all  $s \in \text{Act}^*$ . In this case, the definitions for  $\preceq_{\text{must}}$  (Definition 2.5) and  $\preceq_{\text{unc}}^{\tilde{\mathbb{I}}}$  (Definition 3.7) coincide. Formally, we show that  $p \preceq_{\text{must}} q$  implies  $p \preceq_{\text{unc}}^{\mathbb{I}} q$ . Assume  $p \Downarrow [s]_D$  for  $s \in \text{Act}^*$ . Then

- $p \Downarrow t$  for all  $t \in [s]_D$ . Since  $p \preceq_{\text{must}} q$ , we know that  $p \Downarrow t$  implies  $q \Downarrow t$  for all  $t \in [s]_D$ . Consequently,  $q \Downarrow [s]_D$ .
- Assume  $(p \text{ after } [s]_D) \text{ MUST } L$  for any  $L \subseteq \text{Act}$ . Then,  $(p \text{ after } t) \text{ MUST } L$  for all  $t \in [s]_D$ . Since  $p \preceq_{\text{must}} q$ ,  $(q \text{ after } t) \text{ MUST } L$  for all  $t \in [s]_D$ . Hence,  $(q \text{ after } [s]_D) \text{ MUST } L$ .

□

The converse of Lemma 5.1 and Proposition 5.2 do not hold. Consider the processes  $p = a.b + a + b$  and  $q = b.a + a + b$ . It has been shown, in Example 3.14, that we have  $p \sqsubseteq_{\text{unc}}^{\{\{a\}, \{b\}\}} q$ . Nonetheless, it is easy to check that  $p \not\sqsubseteq_{\text{must}} q$  (i.e.,  $p \not\sqsubseteq_{\text{unc}}^{\{\text{Act}\}} q$ ) by using  $o = \bar{b}.(\tau.1 + \bar{a}.0)$  as observer.

We also have that the individualistic preorder is coarser than the uncoordinated one.

**Proposition 5.3.** *Let  $\mathbb{I}$  be an interface. Then,  $p \sqsubseteq_{\text{unc}}^{\mathbb{I}} q$  implies  $p \sqsubseteq_{\text{ind}}^{\mathbb{I}} q$ .*

*Proof.* Let  $D$  be the dependency relation induced by  $\mathbb{I}$ . We first note that  $[t]_D \subseteq [[s]]_I$  for all  $t \in [[s]]_I$  and  $I \in \mathbb{I}$  because every two strings in the same Mazurkiewicz trace have the same symbols. Then, assume  $p \Downarrow [[s]]_I$  for  $s \in \text{Act}^*$ . Consequently,

- $p \Downarrow t$  for all  $t \in [[s]]_I$ . Since  $[[t]]_I = [[s]]_I$  and  $[t]_D \subseteq [[s]]_I$ ,  $p \Downarrow t'$  for all  $t' \in [t]_D$  and  $t \in [[s]]_I$ . Moreover,  $p \preceq_{\text{unc}}^{\mathbb{I}} q$  implies  $q \Downarrow t'$  for all  $t' \in [t]_D$  and  $t \in [[s]]_I$ . Consequently,  $q \Downarrow [[s]]_I$ .
- Assume  $(p \text{ after } [[s]]_I) \text{ MUST } L \cup (\text{Act} \setminus I)$  with  $L \subseteq I$ . Then,  $(p \upharpoonright I \text{ after } t \upharpoonright I) \text{ MUST } L$ , for all  $t \in [[s]]_I$ . Then  $p \preceq_{\text{unc}}^{\mathbb{I}} q$  implies  $(q \upharpoonright I \text{ after } t' \upharpoonright I) \text{ MUST } L$  for all  $t' \in [t]_D$  and  $t \in [[s]]_I$ .

□



The converse does not hold, i.e.,  $p \sqsubseteq_{\text{ind}}^{\mathbb{I}} q$  does not imply  $q \sqsubseteq_{\text{unc}}^{\mathbb{I}} p$ . Indeed, we have that  $a.b \sqsubseteq_{\text{ind}}^{\{\{a\},\{b\}\}} b.a$  (Example 4.10) but  $a.b \not\sqsubseteq_{\text{unc}}^{\{\{a\},\{b\}\}} b.a$  (Example 3.14).

## 6. MULTIPARTY TESTING AT WORK

In this section we describe a prototype implementation for checking testing preorders among finite processes and use it for analysing a scenario involving a replicated data store and comparing different alternative policies for guaranteeing data consistency.

**6.1. Implementation in Prolog.** To provide the Prolog implementation of the new testing preorders, we rely on their alternative characterisations in term of traces. The actual implementation is restricted to the finite fragment of our specification language (Sequential CCS) and is available from <http://lafhis.dc.uba.ar/users/~hmelgratti/multiTesting.pl>.

Processes are represented as functional terms built-up from the constants 0 and 1, the unary operator  $\sim$  (output actions), and the binary functions  $*$  (prefix) and  $+$  (choice). The operational semantics of finite CCS processes is given by the ternary predicate `red(P,L,Q)`, which is defined in one-to-one correspondence with the inference rules for finite processes (i.e., those rules that do not involve recursive processes) in Definition 2.1. The corresponding Prolog predicates are the following.

```
red(1, tick, 0).
red(L * P, L, P).
red(P + _, L, P1) :- red(P, L, P1).
red(_ + Q, L, Q1) :- red(Q, L, Q1).
```

Now, by building on the predicate `red(.,.,.)`, we inductively define the ternary (*weak reduction*) relation  $P \xRightarrow{S} Q$  as the predicate `wred(P, S, Q)` below.

```
1 wred(P, [], P).
2 wred(P, [L|S], Q) :- red(P, L, R), L \= tau, wred(R, S, Q).
3 wred(P, S, Q) :- red(P, tau, R), wred(R, S, Q).
```

The rules above respectively stand for  $P \Rightarrow P$  (line 1);  $P \xRightarrow{LS} Q$  if  $L \neq \tau$  and  $P \xrightarrow{L} R$  and  $R \xRightarrow{S} Q$  (line 2); and  $P \xRightarrow{S} Q$  if  $P \xrightarrow{\tau} R$  and  $R \xRightarrow{S} Q$  (line 3).

Then, the *set of traces from*  $P$ ,  $S = \text{str}(P)$ , is defined as follows.

```
1 tr(P, T) :- wred(P, T, _).
2 str(P, S) :- setof(T, tr(P,T), S).
```

The set containing the residuals of a process  $P$  after the execution of a sequence of actions  $T$  is defined by the following two rules

```
1 after(P, T, []) :- not(wred(P, T, _)), !.
2 after(P, T, Qs) :- setof(Q, wred(P,T,Q), Qs).
```

The first rule states that  $P$  after  $T = \emptyset$  when  $P$  does not have  $T$  as one of its traces, while the second one handles the case in which  $T$  is a trace of  $P$ . The predicate `after(.,.,.)` is implemented with two rules because `setof(Q, wred(P,T,Q), Qs)` fails when the goal `wred(P,T,Q)` does not have any solution.

The predicate  $P \text{ MUST } L$  of Definition 2.4 is inductively implemented by the following rules.

```

1 must([], _).
2 must([P|Ps], L) :-
3   member(A, L), wred(P, [A], _), !, must(Ps, L).

```

Line 1 stands for the base case, i.e.,  $\emptyset \text{ MUST } L$  for any  $L$ . Differently, Line 2 states that for a non empty set of processes  $\{P\} \cup Ps$ , it should be the case that there exists some action  $A \in L$  such that  $P \xRightarrow{A}$  and  $Ps \text{ MUST } L$ .

Since we are considering the finite fragment of the calculus, we do not need to consider the convergence predicate  $\Downarrow_s$ , which trivially holds for finite processes and finite strings.

We have now all the ingredients needed for the definition of  $\preceq_{\text{must}}$ . We remark that we find useful to state  $\preceq_{\text{must}}$  in terms of  $\not\preceq_{\text{must}}$ , which could provide us with witnesses that explain why two particular processes are not in  $\preceq_{\text{must}}$  relation. We implement  $\not\preceq_{\text{must}}$  as the quaternary predicate `notleqmust(P,Q,S,L)` meaning that  $P \not\preceq_{\text{must}} Q$  because  $(P \text{ after } S) \text{ MUST } L$  but  $(Q \text{ after } S) \text{ MUST } L$ .

```

1 notleqmust(P, Q, S, L):-
2   str(P+Q, Ss), member(S, Ss),
3   after(P, S, Ps), after(Q, S, Qs),
4   n(P+Q, As), subseq(L, As),
5   must(Ps, L), not(must(Qs, L)).
6
7 leqmust(P,Q) :- not(notleqmust(P,Q,_,_)).

```

Line 2 states that we only consider the set `Ss` of traces that are either traces of `P` or `Q` and disregard any other trace because the residuals for both `P` and `Q` are empty in those cases, and hence uninteresting. When defining must-sets, it is useless to consider actions that are not in the alphabet of the processes<sup>1</sup>. Therefore, line 4 states that we only consider subsets `L` of the names occurring in either `P` or `Q`. Then, in order to show that `P` and `Q` are not in  $\preceq_{\text{must}}$  relation, we search for a set `L` that is a must-set of the residuals of `P` after `S` (i.e., `must(Ps, L)`) but not of the residuals of `Q` after `S` (line 5). Finally, the predicate  $\preceq_{\text{must}}$  is just defined as the negation of  $\not\preceq_{\text{must}}$  (line 7).

As an example of use of the `notleqmust(_,_,_,_)` predicate, we can use it to show that neither  $\mathbf{0} \sqsubseteq_{\text{must}} \tau.a + \tau.b$  nor  $\tau.a + \tau.b \sqsubseteq_{\text{must}} \mathbf{0}$  hold.

In fact, the following query

```

1 ?- notleqmust(0, tau * a * 0 + tau * b * 0, S, L).

```

has several solutions, among which we have `S = [a]`, `L = []`.

Similarly,

```

1 ?- notleqmust(tau * a * 0 + tau * b * 0, 0, S, L).

```

has `S = []`, `L = [a, b]` among its solutions.

The implementation for the uncoordinated and individualistic preorders follows analogously. First, we generalise the definition of residuals to consider a set of traces instead of

<sup>1</sup>The definition of predicate `n(P,As)`, which computes the alphabet of `P`, has been omitted because it is straightforward.

just a trace. This is done just by collecting all the residuals of the process for each trace in the set. We use the ternary predicate `afterC(, , )` defined as follows.

```

1 afterC(, [], []).
2 afterC(P, [X|Xs], Ps) :- after(P, X, P1s), afterC(P, Xs, P2s),
3   union(P1s, P2s, Ps).
```

In addition, we use two auxiliary predicates: `independence(I, Ind)`, which computes the independence relation `Ind` induced by an interface `I`; and `mazurkiewicz(Ind, S, CT)`, which takes an independence relation `Ind`, a set of traces `S` belonging to the same equivalence class, and generates the complete set of traces `CT` in that equivalence class. We omit here their definition because are straightforward and not interesting.

As for the classical must preorder, we implement  $\preceq_{\text{unc}}^{\mathbb{I}}$  in terms of  $\not\preceq_{\text{unc}}^{\mathbb{I}}$ , which is defined by the predicate `notlequnc(P, Q, I, T, L)`, in which the additional parameter `I` stands for the interface. Its definition is below.

```

1 notlequnc(P, Q, I, T, L) :-
2   independence(I, Ind),
3   str(P+Q, Ts), member(T, Ts),
4   mazurkiewicz(Ind, [T], CT), afterC(P, CT, P1), afterC(Q, CT, Q1),
5   member(P1, I), subseq(L, P1),
6   must(P1, L), not(must(Q1, L)).
7
8 lequnc(P, Q, I) :- not(notlequnc(P, Q, I, _, _)).
```

The differences with respect to the definition of `notlequnc(, , , , )` are the following:

- we compute the independence relation `Ind` induced by the interface `I` (line 2);
- residuals are obtained for each equivalence class of a trace (line 4) (instead of just a trace);
- must-sets are built with actions in just one part of the interface (line 5).

Then, we can check, e.g., that  $\tau.a + \tau.b \sqsubseteq_{\text{unc}}^{\mathbb{I}} 0$  for  $\mathbb{I} = \{\{a\}, \{b\}\}$  by executing the query

```

1 ?- notlequnc(tau * a * 0 + tau * b * 0, 0, [[a], [b]], T, L).
```

which does not have any solutions.

Also, we can test that  $a.b \not\sqsubseteq_{\text{unc}}^{\mathbb{I}} b.a$  for  $\mathbb{I} = \{\{a\}, \{b\}\}$ , because the query

```

1 ?- notlequnc(b * a * 0, a * b * 0, [[a], [b]], T, L).
```

has several solutions, among which we have `T = [], L = [b]`.

The implementation of the individualistic preorder consists in the definition of an analogous predicate `notleqind(P, Q, I, T, L)`, which considers the equivalence classes of filtered traces instead of the Mazurkiewicz ones. It is defined as follows.

```

1 notleqind(P, Q, I, T, L1) :-
2   str(P+Q, Ts), member(T, Ts), member(P1, I),
3   filtered(T, P1, Ts, CT), afterC(P, CT, P1), afterC(Q, CT, Q1),
4   complement(I, P1, C), subseq(L1, P1), append(L1, C, L),
5   must(P1, L), not(must(Q1, L)).
6
```

```
7 leqind(P,Q,I) :- not(notlequnc(P,Q,I,_,_)).
```

In this case the variable **CT** in line 3 stands for the (relevant part of the) equivalence class of the trace **T**. Since the equivalence classes for the filtered case are all infinite and, hence, cannot be computed completely, the predicate **filtered(T,PI,Ts,CT)** simply generates the traces in the equivalence class of **T** that are also traces of at least one of the two processes under comparison (note that the residuals are empty for both processes in the remaining cases, and hence irrelevant). The definition of **filtered(T,PI,Ts,CT)** takes a part of the interface **PI** and a set of traces **Ts**, and returns **CT** which contains the set of traces in **Ts** whose projection over **PI** coincides with the projection of **T**. Note that **Ts** in line 3 corresponds to the traces in either **P** or **Q** (line 2). The remaining difference concerns to the generation of must-sets (line 4). In this case, each candidate must-set **L** contains a subset **L1** of the part of the interface under analysis **PI** and the set **C** containing all actions in the interface **I** that are not in **PI** (this set is computed by the predicate **complement(I, PI, C)**, whose definition has been omitted).

We can use this predicate to check, e.g., that  $a.b \sqsubseteq_{\text{ind}}^{\mathbb{I}} b.a$  for  $\mathbb{I} = \{\{a\}, \{b\}\}$  by executing the query

```
1 ?- notlequnc( a * b * 0, b * a * 0, [[a],[b]], T,L).
```

which does not have any solution.

Also, we may check that  $a.b \sqsubseteq_{\text{ind}}^{\mathbb{I}} b.a$  does not hold when  $\mathbb{I} = \{\{a,b\}\}$  because the query

```
1 ?- notlequnc( a * b * 0, b * a * 0, [[a,b]], T,L).
```

has several solutions, e.g.,  $T = []$ ,  $L = [a]$ .

We now illustrate the use of the introduced preorders and of our prototype implementation in a larger scenario.

**6.2. A case study: Cassandra.** Distributed, non-relational databases such as Dynamo [DHJ<sup>+</sup>07] and Cassandra [LM10] provide highly available storage by replicating data and relaxing consistency guarantees. Such databases store key-value pairs that can be accessed by using two operations: **get** to retrieve the value associated with a key, and **put** to store the value of a particular key. A client issuing an operation interacts with the closest server, which plays the role of a coordinator and mediates between the client and the replicas to complete the client request. Each client request is associated with a consistency level, which specifies the degree of consistency required over data. For a **put** operation, the consistency level states the number of replicas that must be written before sending an acknowledgement to the client. Similarly, the consistency level of a **get** operation specifies the number of replicas that must reply to the read request before returning the data to the client. Cassandra provides several consistency levels; for instance, an operation may request to be performed over just **ONE** or **TWO** replicas, or over the majority of the replicas (i.e., **QUORUM**) or over **ALL** the replicas. Consequently, depending on the consistency level required by the client, the coordinator chooses the replicas to contact.

We will now describe the behaviour of a node acting as coordinator in a configuration that involves two additional replicas. Then we will introduce alternative policies the coordinator might want to use when reacting to users request and will discuss their relationships.

For simplicity reasons, we just illustrate the protocol for processing the operation **get** and abstract away from the values exchanged during the communication (the **put** operation is analogous). The actual protocol for handling a **get** is described below as a CCS process.

$$\begin{aligned}
\text{Coord} &\stackrel{\text{def}}{=} \text{get} . (\tau.\overline{\text{err}} + \tau.\overline{\text{ret}} + \tau.\text{Query}_1 + \tau.\text{Query}_2 + \tau.\text{Query}_{1,2} + \tau.\text{Query}_{2,1}) \\
\text{Query}_i &\stackrel{\text{def}}{=} \overline{\text{read}}_i . (\tau.\overline{\text{err}} + \text{ret}_i . (\tau.\overline{\text{err}} + \tau.\overline{\text{ret}})) \\
\text{Query}_{i,j} &\stackrel{\text{def}}{=} \overline{\text{read}}_i . \overline{\text{read}}_j . (\tau.\overline{\text{err}} + \text{Ans}_{i,j} + \text{Ans}_{j,i}) \\
\text{Ans}_{i,j} &\stackrel{\text{def}}{=} \text{ret}_i . (\tau.\overline{\text{err}} + \tau.\overline{\text{ret}} + \text{ret}_j . (\tau.\overline{\text{ret}} + \tau.\overline{\text{err}}))
\end{aligned}$$

As stated in **Coord**, the coordinator after receiving the request **get** internally decides to either:

- reply to the client with the error message **err**, e.g., when the available nodes are not enough to guarantee the requested consistency level; or
- return the requested information by using just local information (message **ret**); or
- retrieve information by contacting just one of the additional replicas following the protocol defined by **Query<sub>i</sub>**; or
- retrieve information from both replicas, following the protocol defined by **Query<sub>i,j</sub>**.

The protocol followed by the coordinator when contacting replica *i* is modelled by process **Query<sub>i</sub>**: The coordinator sends a read request over the channel **read<sub>i</sub>** and awaits an answer on channel **ret<sub>i</sub>**, however it may internally decide not to wait for the answer from the replica and send an error to the client (e.g., in a timeout expires). When the coordinator receives the response from the replica, it may return the requested information to the client or signal an error (e.g., when the consistency level cannot be satisfied by the current state of the replicas).

The protocol followed by the coordinator when contacting both replicas is modelled by process **Query<sub>i,j</sub>**: When awaiting for their responses, the coordinator may internally decide to reply to the client before or after receiving any of the two answers.

Any equation **name**  $\stackrel{\text{def}}{=} \text{proc}$  above can be defined in Prolog by using the predicate **proc(name, proc)** as shown below.

```

1 proc(coord, get*(tau~err*0 + tau~ret*0 + tau*Query1 + tau*Query2
2           + tau*Query12 + tau*Query21))
3 :- proc(query(1), Query1), proc(query(2), Query2),
4     proc(query(1,2), Query12), proc(query(2,1), Query21).
5
6
7 proc(query(I), ~read(I)*(tau~err*0 + ret(I)*(tau~err*0 + tau~ret*0))).
8
9 proc(query(I,J), ~read(I)*~read(J)*(tau~err*0 + AnsIJ + AnsJI))
10 :- proc(ans(I,J), AnsIJ), proc(ans(J,I), AnsJI).
11
12 proc(ans(I,J), ret(I)*(tau~ret*0 + tau~err*0
13           + ret(J)*(tau~ret*0 + tau~err*0))).

```

A possible implementation of **Coord** may only provide the part of the protocol that always contact the two additional replicas regardless of the information and consistency

level requested by the client. Such implementation can be described as follows.

$$\text{Coord}_1 \stackrel{\text{def}}{=} \text{get.Query}_{1,2}$$

where  $\text{Query}_{1,2}$  is as before. This defining equation is implemented in Prolog as follows:

```
1 proc(coord1, get*Query12) :- proc(query(1,2), Query12).
```

We can check that  $\text{Coord} \sqsubseteq_{\text{must}} \text{Coord}_1$  by performing the query

```
1 ?:- proc(coord, Coord), proc(coord,Coord1), leqmust(Coord,Coord1).
```

An alternative implementation of  $\text{Coord}$  may decide to communicate an error to the client but still accept responses from the replicas after this interaction. This feature allows the coordinator to update its local state with information that can be used when answering future requests. Such implementation can be described as follows:

$$\begin{aligned} \text{Coord}_2 &\stackrel{\text{def}}{=} \text{get.read}_1.\text{read}_2.(\tau.\overline{\text{err}}.\text{ret}_1.\text{ret}_2 + \text{Wait}_{1,2} + \text{Wait}_{2,1}) \\ \text{Wait}_{i,j} &\stackrel{\text{def}}{=} \text{ret}_i.(\tau.\overline{\text{ret}}.\text{ret}_j + \tau.\overline{\text{err}}.\text{ret}_j + \text{ret}_j.(\tau.\overline{\text{ret}} + \tau.\overline{\text{err}})) \end{aligned}$$

Note that  $\text{Coord}_2$  accepts responses from the replicas even after it has replied to the client. As for  $\text{Coord}$ , the definition of  $\text{Coord}_2$  in Prolog is straightforward (and omitted here). When considering the classical must testing preorder, it holds that  $\text{Coord} \not\sqsubseteq_{\text{must}} \text{Coord}_2$ . However, as far as the behaviour of the client and the replicas is concerned, the implementation of  $\text{Coord}_2$  is harmless. In fact, we can prove that  $\text{Coord} \sqsubseteq_{\text{unc}}^{\mathbb{I}} \text{Coord}_2$  when

$$\mathbb{I} = \{\{\text{get}, \text{ret}, \text{err}\}, \{\text{get}, \text{read}_1, \text{ret}_1\}, \{\text{get}, \text{read}_2, \text{ret}_2\}\}$$

For convenience, when querying the program we add the following definition rule for the interface.

```
1 int([[get, ~ret, ~err], [~read(1), ret(1)], [~read(2), ret(2)]]).
```

and then query the program as follows:

```
1 ?- proc(coord, C), proc(coord2,C2), int(I), notlequnc(C,C2,I,_,_).
```

The query above has no solutions, and hence  $\text{Coord} \sqsubseteq_{\text{unc}}^{\mathbb{I}} \text{Coord}_2$ . Similarly, it can be proved that  $\text{Coord}_1 \sqsubseteq_{\text{unc}}^{\mathbb{I}} \text{Coord}_2$ . On the contrary, it can be checked the neither  $\text{Coord}_2 \sqsubseteq_{\text{unc}}^{\mathbb{I}} \text{Coord}$  nor  $\text{Coord}_2 \sqsubseteq_{\text{unc}}^{\mathbb{I}} \text{Coord}_1$ . For instance, the query

```
1 ?- proc(coord1, C), proc(coord2,C2), int(I), notlequnc(C2,C1,I,S,L).
```

has several solutions, e.g.,:

- $S = [\text{get}, \sim\text{read}(1), \sim\text{read}(2), \text{ret}(1), \sim\text{err}], L = [\text{ret}(2)],$
- $S = [\text{get}, \sim\text{read}(1), \sim\text{read}(2), \text{ret}(2), \sim\text{err}], L = [\text{ret}(1)],$
- $S = [\text{get}, \sim\text{read}(1), \sim\text{read}(2), \sim\text{err}], L = [\text{ret}(1)],$
- $S = [\text{get}, \sim\text{read}(1), \sim\text{read}(2), \sim\text{err}, \text{ret}(1)], L = [\text{ret}(2)].$

All of them show that  $\text{Coord}_2$  is able to accept an answer from a replica even after signaling an error to the client, while  $\text{Coord}_1$  is not. Consequently, a replica may distinguish the behaviours of the different implementations: when interacting with  $\text{Coord}_1$ , a replica  $i$  may discover that the coordinator has sent the message  $\overline{\text{err}}$  to the client because the interaction  $\overline{\text{ret}}_i$  cannot not take place.

We now consider a variant of  $\text{Coord}_2$  that chooses a different order for contacting replicas, defined as follows:

$$\text{Coord}_3 \stackrel{\text{def}}{=} \text{get}.\overline{\text{read}}_2.\overline{\text{read}}_1.(\tau.\overline{\text{err}}.\text{ret}_1.\text{ret}_2 + \text{Wait}_{1,2} + \text{Wait}_{2,1})$$

where  $\text{Wait}_{i,j}$  is defined as before. The only difference between  $\text{Coord}_2$  and  $\text{Coord}_3$  is the order in which  $\overline{\text{read}}_1$  and  $\overline{\text{read}}_2$  are executed.

We have that  $\text{Coord}_2$  and  $\text{Coord}_3$  are still distinguishable in the uncoordinated preorder. For instance, the query

```
1 ?- proc(coord2, C2), proc(coord3, C3), int(I), notlegunc(C2, C3, I, S, L).
```

has among its solutions the following one:

```
1 S = [get], L = [~read(1)]
```

showing that  $\text{Coord}_2 \not\sqsubseteq_{\text{unc}}^{\mathbb{I}} \text{Coord}_3$ . The test associated with the above witness is built by preventing the interaction with the replica 2 (i.e., when the communication over  $\text{read}_2$  is not enabled). However, if the interaction with the replicas is guaranteed, both implementations should be deemed as indistinguishable. In fact,  $\text{Coord}_2$  and  $\text{Coord}_3$  are indistinguishable in the individualistic preorder. We remark, however, that  $\text{Coord}_1$  is still not equivalent to either  $\text{Coord}_2$  or  $\text{Coord}_3$ . For instance, the pair

```
1 S = [get, ~read(1)], L = [~ret(1)]
```

witnesses the fact that  $\text{Coord}_3 \not\sqsubseteq_{\text{ind}}^{\mathbb{I}} \text{Coord}_1$ . In fact, while  $\text{Coord}_3$  ensures that it will always receive the reply from the replica 1 after sending the request  $\text{read}_1$ . This is not the case for  $\text{Coord}_1$ , which may refuse to communicate over  $\text{read}_1$ , e.g., after an internal timeout.

## 7. CONCLUSIONS AND RELATED WORKS

In this paper we have explored different relaxations of the must testing preorder tailored to define new behavioural relations that, in the framework of Service Oriented Computing, are better suited to study compliance between contracts exposed by clients and servers interacting via synchronous binary communication primitives.

In particular, we have considered two different scenarios in which contexts of a service are represented by processes with distributed control. The first variant, that we called uncoordinated preorder, corresponds to multiparty contexts without runtime communication between peers but with the possibility of one peer to block another if it does not perform the expected action. Indeed, the observations at the basis of our experiments are designed with the assumption that the users of a service interact only via dedicated ports but might be influenced by the fact that other partners do not perform the expected actions. The second preorder we introduced is called individualistic preorder. It accounts for partners that are completely independent from the behaviour of the other ones. Indeed, from a viewpoint of a client, actions by other clients are considered unobservable.

We have shown that the discriminating power of the induced equivalences decreases as observers become weaker; and thus that the individualistic preorder is coarser than the uncoordinated preorder which in turn is coarser than the classical testing preorder. As future work we plan to consider different "real life" scenarios and to assess the impact of the different assumptions at the basis of the two new preorders and the identifications/orderings

they induce. We plan also to perform further studies to get a fuller account, possibly via axiomatisations, of their discriminating power. In the near future, we will also consider the impact of our testing framework on calculi based on asynchronous interactions.

Several variants of the must testing preorder, contract compliance and sub-contract relation have been developed in the literature to deal with different aspects of services compositions, such as buffered asynchronous communication [BZ08, Pad10, MYH09], fairness [Pad11], peer-interaction [BH13]. We have however to remark that these approaches deal with aspects that are somehow orthogonal to the discriminating power of the distributed tests analysed in this work. Our preorders have some similarities with those relying on buffered communications in that both aim at guaranteeing that actions performed by independent peers can be reordered, but we rely on synchronous communication and, hence, message reordering is not obtained by swapping buffered messages but by relying on more generous observers. As mentioned above, we have left the study of distributed tests under asynchronous communication as a future work. However, we would like to remark that, even the uncoordinated and the individualistic preorders are different from those in [BZ08, Pad10, MYH09] that permit explicit action reordering. The paradigmatic example is the equivalence  $a.c + b.d \approx_{\text{ind}}^{\{a,b\},\{c,d\}} a.d + b.c$ , which does not hold for any of the preorders with buffered communication. The main reason is that, even in presence of buffered communication, the causality, e.g., between  $a$  and  $c$  is always observed.

## ACKNOWLEDGMENTS

We are very grateful to Marzia Buscemi for interesting discussions during early stages of this work. We thank the anonymous reviewers of CONCUR and TGC 2015 for their careful reading of our manuscript and their many insightful comments and suggestions. The work has been partially supported by CONICET (under project PIP 11220130100148CO), UBA (under project UBACYT 20020130200092BA), and MIUR under PRIN project CINA.

## REFERENCES

- [BBO12] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In *ACM SIGPLAN Notices*, volume 47, pages 191–202. ACM, 2012.
- [BH13] Giovanni Bernardi and Matthew Hennessy. Mutually testing processes. In *CONCUR 2013–Concurrency Theory*, pages 61–75. Springer, 2013.
- [BZ07] Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *SC*, volume 4829 of *Lect. Notes in Comput. Sci.*, pages 34–50. Springer Verlag, 2007.
- [BZ08] Mario Bravetti and Gianluigi Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundamenta Informaticae*, 89(4):451–478, 2008.
- [CGP08] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *POPL*, pages 261–272, 2008.
- [CGP09] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.
- [DH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.
- [DHJ<sup>+</sup>07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: amazon’s highly available key-value store. In *Proceedings of SOSP’2007*, pages 205–220. ACM, 2007.



- [HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 122–138. Springer Verlag, 1998.
- [LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.
- [LP07] Cosimo Laneve and Luca Padovani. The Must preorder revisited. In *CONCUR*, volume 4703 of *Lect. Notes in Comput. Sci.*, pages 212–225, 2007.
- [Maz86] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.
- [Maz95] Antoni W. Mazurkiewicz. Introduction to trace theory. *The Book of Traces*, pages 3–41, 1995.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
- [MYH09] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *Programming Languages and Systems*, pages 316–332. Springer, 2009.
- [Pad10] Luca Padovani. Contract-based discovery of web services modulo simple orchestrators. *Theoretical Computer Science*, 411(37):3328–3347, 2010.
- [Pad11] Luca Padovani. Fair subtyping for multi-party session types. In *Coordination Models and Languages*, pages 127–141. Springer, 2011.
- [THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *Lect. Notes in Comput. Sci.*, pages 398–413. Springer Verlag, 1994.